1. Factores del coste del Ciclo de Vida del Software
2. Técnicas del Mantenimiento del Software
3. ¿Qué es la Ingeniería Inversa?
4. Ingeniería Inversa de Procesos
5. Ingeniería Inversa de Datos
6. Técnicas de la Ingeniería Inversa de Datos
7. Ejemplo de Ingeniería Inversa de Datos
8. Ingeniería Inversa de Interfaces de Usuario
9. ¿Qué es Reingeniería del Software?
10. Procesos involucrados en la Reingeniería del Software
11. Valoraciones sobre la Reingeniería del Software
12. ¿Qué es Reestructuración del Software?
13. ¿Qué es Refactoring?

Factores del coste del Ciclo de Vida del Software

El siguiente gráfico, puede mostrar la distribución del coste del ciclo de vida:

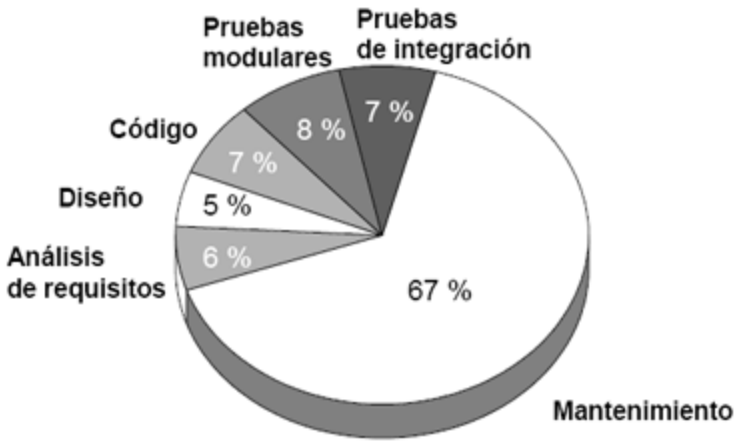


Figura 1. Distribución del coste del ciclo de vida

El mantenimiento del software es una importante tarea que habitualmente requiere entre el 70% y 80% del coste del ciclo de vida del producto. Esto es debido a múltiples factores, entre los que podemos encontrar:

- Inexistencia de métodos, técnicas y herramientas que puedan proporcionar una solución global al mantenimiento.
- La complejidad de los sistemas se incrementa paulatinamente por la realización de continuas modificaciones.
- La documentación del sistema es defectuosa e inexistente.
- Se considera el mantenimiento como una actividad poco creativa, a diferencia del desarrollo.
- Las actividades de mantenimiento se suelen realizar bajo presión de tiempo.
- Poca participación del usuario durante el desarrollo del sistema.

Las actuaciones comunes para mantener la operatividad del software son:

- Corrección de defectos en el software.
- Creación de nuevas funcionalidades en el software por nuevos requisitos de usuario.

- Mejora de la funcionalidad y del rendimiento.

La distribución del tiempo en tareas de mantenimiento se muestra en el siguiente gráfico:

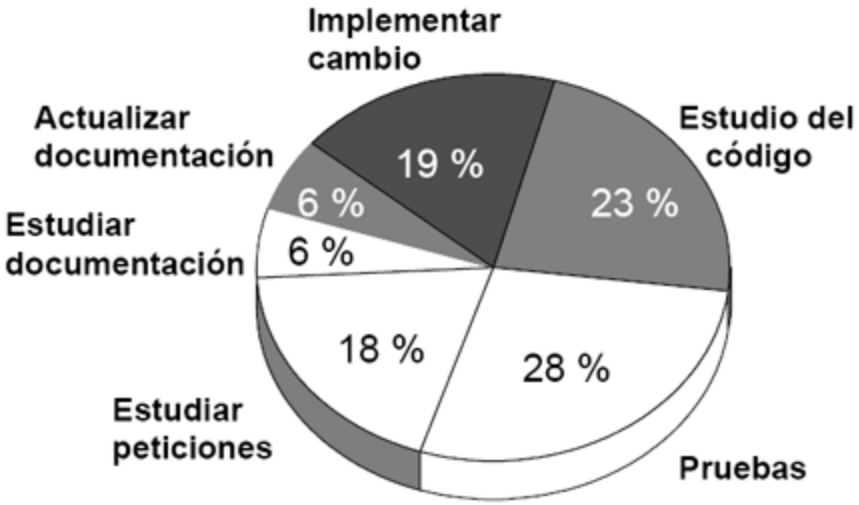


Figura 2. Distribución del tiempo en tareas de Mantenimiento

El error que sucede habitualmente es que se dedica escaso tiempo al mantenimiento del software. Se planifica más tiempo para realizar desarrollos (programar) que para el mantenimiento, de ahí, que luego haya errores de costes y de planificación. También es importante el factor de la documentación. Esta siempre tiene que estar actualizada para posibles cambios en el futuro.

Técnicas del Mantenimiento del Software

Dentro de la ingeniería del software se proporcionan soluciones técnicas que permiten abordar el mantenimiento de manera que su impacto en coste dentro del ciclo de vida sea menor. Las soluciones técnicas pueden ser de tres tipos:

1. Ingeniería inversa: Análisis de un sistema para identificar sus componentes y las relaciones entre ellos, así como para crear representaciones del sistema en otra forma o en un nivel de abstracción más elevado.
2. Reingeniería: Modificación de un producto software, o de ciertos componentes, usando para el análisis del sistema existente técnicas de ingeniería inversa y, para la etapa de reconstrucción, herramientas de ingeniería directa, de tal manera que se oriente este cambio hacia mayores niveles de facilidad en cuanto a mantenimiento, reutilización, comprensión o evolución.
3. Reestructuración del software: Cambio de representación de un producto software, pero dentro del mismo nivel de abstracción.

El objetivos de estas técnicas es proporcionar métodos para reconstruir el software, ya sea reprogramándolo, redocumentándolo, rediseñándolo, o rehaciendo alguna/s característica/s del producto. La diferencia entre las soluciones descritas radica en cuál es el origen y cuál es el destino de las mismas (producto inicial y/o producto final).

Gráficamente, estas tres soluciones técnicas se enmarcan en el ciclo de vida de la siguiente manera:

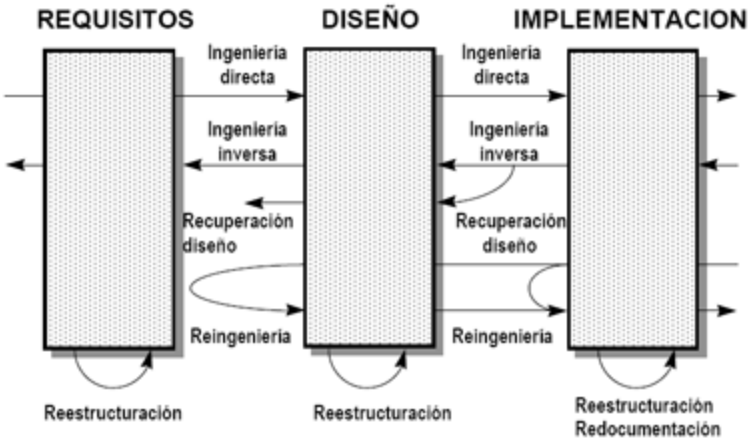


Figura 1. Relaciones entre los términos asociados con la Reingeniería.

La Ingeniería directa corresponde al desarrollo del software tradicional. La Ingeniería Inversa es el proceso de análisis de un sistema para identificar sus componentes e interrelaciones y crear representaciones del sistema en otra forma o a un nivel más alto de abstracción. La Reingeniería es el examen y la alteración de un sistema para reconstruirlo de una nueva forma y la subsiguiente implementación de esta nueva forma. La Reestructuración es la modificación del software para hacerlo más fácil de entender y cambiar.

La reingeniería hace referencia a un ciclo, esto es, se aplican técnicas de ingeniería inversa para conseguir representaciones de mayor abstracción del producto y sobre ellas se aplican técnicas de ingeniería directa para rediseñar o reimplementar el producto.

Cualquiera de estas técnicas se puede aplicar a lo largo de todas las fases del ciclo de vida o bien entre algunas de sus fases.

También existen otras tecnologías, como por ejemplo:

- La remodularización: consiste en cambiar la estructura modular de un sistema de forma que se obtenga una nueva estructura siguiendo los principios del diseño estructurado.
- Análisis de la facilidad de mantenimiento: normalmente la mayor parte del mantenimiento se centra relativamente en unos pocos módulos del

sistema.

- Visualización: el proceso más antiguo para la comprensión del software.
- Análisis y mediciones: son importantes tecnologías que estudian ciertas propiedades de los programas.

¿Qué es la Ingeniería Inversa?

La ingeniería inversa se ha definido como el proceso de construir especificaciones de un mayor nivel de abstracción partiendo del código fuente de un sistema software o cualquier otro producto (se puede utilizar como punto de partida cualquier otro elemento de diseño, etc.).

Estas especificaciones pueden volver ser utilizadas para construir una nueva implementación del sistema utilizando, por ejemplo, técnicas de ingeniería directa.

Beneficios de Ingeniería Inversa

La aplicación de ingeniería inversa nunca cambia la funcionalidad del software sino que permite obtener productos que indican cómo se ha construido el mismo. Se realiza permite obtener los siguientes beneficios:

- Reducir la complejidad del sistema: al intentar comprender el software se facilita su mantenimiento y la complejidad existente disminuye.
- Generar diferentes alternativas: del punto de partida del proceso, principalmente código fuente, se generan representaciones gráficas lo que facilita su comprensión.
- Recuperar y/o actualizar la información perdida (cambios que no se documentaron en su momento): en la evolución del sistema se realizan cambios que no se suele actualizar en las representaciones de nivel de abstracción más alto, para lo cual se utiliza la recuperación de diseño.
- Detectar efectos laterales: los cambios que se puedan realizar en un sistema puede conducirnos a que surjan efectos no deseados, esta serie de anomalías puede ser detectados por la ingeniería inversa.
- Facilitar la reutilización: por medio de la ingeniería inversa se pueden detectar componentes de posible reutilización de sistemas existentes, pudiendo aumentar la productividad, reducir los costes y los riesgos de mantenimiento.

La finalidad de la ingeniería inversa es la de desentrañar los misterios y secretos de los sistemas en uso a partir del código. Para ello, se emplean

una serie de herramientas que extraen información de los datos, procedimientos y arquitectura del sistema existente.

Tipos de Ingeniería Inversa

La ingeniería inversa puede ser de varios tipos:

- Ingeniería inversa de datos: Se aplica sobre algún código de bases datos (aplicación, código SQL, etc) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad-relación
- Ingeniería inversa de lógica o de proceso: Cuando la ingeniería inversa se aplica sobre código de un programa para averiguar su lógica o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos.
- Ingeniería inversa de interfaces de usuario: Se aplica con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.

Herramientas para la Ingeniería Inversa

Los Depuradores

Un depurador es un programa que se utiliza para controlar otros programas. Permite avanzar paso a paso por el código, rastrear fallos, establecer puntos de control y observar las variables y el estado de la memoria en un momento dado del programa que se esté depurando. Los depuradores son muy valiosos a la hora de determinar el flujo lógico del programa.

Un punto de ruptura (breakpoint) es una instrucción al depurador que permite parar la ejecución del programa cuando cierta condición se cumpla. Por ejemplo, cuando un programa accede a cierta variable, o llama a cierta función de la API, el depurador puede parar la ejecución del programa.

Algunos depuradores de Windows son:

- OllyDbg → es un potente depurador con un motor de ensamblado y desensamblado integrado. Tiene numerosas otras características incluyendo un precio de 0 $. Muy útil para parcheado, desensamblado y depuración.
- WinDBG → es una pieza de software gratuita de Microsoft que puede ser usada para depuración local en modo usuario, o incluso depuración remota en modo kernel.

Las Herramientas de Inyección de Fallos

Las herramientas que pueden proporcionar entradas malformadas con formato inadecuado a procesos del software objetivo para provocar errores son una clase de herramientas de inserción de fallos. Los errores del programa pueden ser analizados para determinar si los errores existen en el software objetivo. Algunos fallos tienen implicaciones en la seguridad, como los fallos que permiten un acceso directo del asaltante al ordenador principal o red. Hay herramientas de inyección de fallos basados en el anfitrión que funcionan como depuradores y pueden alterar las condiciones del programa para observar los resultados y también están los inyectores basados en redes que manipulan el tráfico de la red para determinar el efecto en el aparato receptor.

Los Desensambladores

Se trata de una herramienta que convierte código máquina en lenguaje ensamblador. El lenguaje ensamblador es una forma legible para los humanos del código máquina. Los desensambladotes revelan que instrucciones máquinas son usadas en el código. El código máquina normalmente es específico para una arquitectura dada del hardware. De forma que los desensambladotes son escritor expresamente para la arquitectura del hardware del software a desensamblar.

Algunos ejemplos de desensambladores son:

- IDA Pro → es un desensamblador profesional extremadamente potente. La parte mala es su elevado precio.
- PE Explorer → es un desensamblador que “se centra en facilidad de uso, claridad y navegación”. No es tan completo como IDA Pro, pero tiene un precio más bajo.
- IDA Pro Freeware 4.1 → se comporta casi como IDA Pro, pero solo desensambla código para procesadores Intel x86 y solo funciona en Windows.
- Bastard Disassembler → es un potente y programable desensamblador para Linux y FreeBSD.
- Ciasdis → esta herramienta basada en Forth permite construir conocimiento sobre un cuerpo de código de manera interactiva e incremental. Es único en que todo el código desensamblado puede ser re-ensamblado exactamente al mismo código.

Los compiladores Inversos o Decompiladores

Un decompilador es una herramienta que transforma código en ensamblador o código máquina en código fuente en lenguaje de alto nivel. También existen decompiladores que transforman lenguae intermedio en código fuente en lenguaje de alto nivel. Estas herramientas son sumamente útiles para determinar la lógica a nivel superior como bucles o declaraciones if-then de los programas que son decompilados. Los decompiladores son parecidos a los desensambladotes pero llevan el proceso un importante paso más allá.

Algunos decompiladores pueden ser:

- DCC Decompiler → es una exacelente perspectiva teórica a la descompilación, pero el descompilador sólo soporta programas MSDOS.
- Boomerang Decompiler Project → es un intento de construir un potente descompilador para varias máquinas y lenguajes.

- Reverse Engineering Compiler (REC) → es un potente “descompilador” que descompila código ensamblador a una representación del código semejante a C. El código está a medio camino entre ensamblador y C, pero es mucho más legible que el ensamblador puro.

Las Herramientas CASE

Las herramientas de ingeniería de sistemas asistida por ordenador (Computer-Aided Systems Engineering – CASE) aplican la tecnología informática a las actividades, las técnicas y las metodologías propias de desarrollo de sistemas para automatizar o apoyar una o más fases del ciclo de vida del desarrollo de sistemas. En el caso de la ingeniería inversa generalmente este tipo de herramientas suelen englobar una o más de las anteriores junto con otras que mejoran el rendimiento y la eficiencia.

Ingeniería Inversa de Procesos

La primera actividad real de la ingeniería inversa comienza con un intento de comprender y posteriormente, extraer las abstracciones de procedimientos representadas por el código fuente. Para comprender las abstracciones de procedimientos, se analiza el código en distintos niveles de abstracción: sistema, programa, componente, configuración y sentencia.

Antes de iniciar el trabajo de ingeniería inversa detallado debe comprenderse totalmente la funcionalidad general de todo el sistema de aplicaciones sobre el que se esta operando. Esto es lo que establece un contexto para un análisis posterior, y proporciona ideas generales acerca de los problemas de interoperabilidad entre aplicaciones dentro del sistema. Así pues, cada uno de los programas de que consta el sistema de aplicaciones representará una abstracción funcional con un elevado nivel de detalle, creándose un diagrama de bloques como representación de la iteración entre estas abstracciones funcionales. Cada uno de los componentes de estos diagramas efectúa una subfunción, y representa una abstracción definida de procedimientos. En cada componente se crea una narrativa de procesamientos. En algunas situaciones ya existen especificaciones de sistema, programa y componente. Cuando ocurre tal cosa, se revisan las especificaciones para preciar si se ajustan al código existente, descartando posibles errores.

Todo se complica cuando se considera el código que reside en el interior del componente. El ingeniero busca las secciones del código que representan las configuraciones genéricas de procedimientos. En casi todos los componentes, existe una sección de código que prepara los datos para su procesamiento (dentro del componente), una sección diferente de código que efectúa el procesamiento y otra sección de código que prepara los resultados del procesamiento para exportarlos de ese componente. En el interior de cada una de estas secciones, se encuentran configuraciones más pequeñas. Por ejemplo, suele producirse una verificación de los datos y una comprobación de los límites dentro de la sección de código que prepara los datos para su procesamiento.

Para los sistemas grandes, la ingeniería inversa suele efectuarse mediante el uso de un enfoque semiautomatizado. Las herramientas CASE se utilizan

para “analizar” la semántica del código existente. La salida de este proceso se pasa entonces a unas herramientas de reestructuración y de ingeniería directa que completarán el proceso de reingeniería.

Cuándo aplicar ingeniería inversa de procesos

Cuando la ingeniería inversa se aplica sobre código de un programa para averiguar su lógica o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos se habla de ingeniería inversa de procesos.

Habitualmente, este tipo de ingeniería inversa se usa para:

- Entender mejor la aplicación y regenerar el código.
- Migrar la aplicación a un nuevo sistema operativo.
- Generar/completar la documentación.
- Comprobar que el código cumple las especificaciones de diseño.

La información extraída son las especificaciones de diseño: se crean modelos de flujo de control, diagramas de diseño, documentos de especificación de diseño, etc. y pudiendo tomar estas especificaciones como nuevo punto de partida para aplicar ingeniería inversa y obtener información a mayor nivel de abstracción.

¿Cómo hacemos la ingeniería inversa de procesos?

A la hora de realizar ingeniería inversa de procesos se suelen seguir los siguientes pasos:

1. Buscamos el programa principal.
2. Ignoramos inicializaciones de variables, etc.
3. Inspeccionamos la primera rutina llamada y la examinamos si es importante.
4. Inspeccionamos las rutinas llamadas por la primera rutina del programa principal, y examinamos aquéllas que nos parecen importantes.
5. Repetimos los pasos 3-4 a lo largo del resto del software.

6. Recopilamos esas rutinas “importantes”, que se llaman componentes funcionales.
7. Asignamos significado a cada componente funcional, esto es (a) explicamos qué hace cada componente funcional en el conjunto del sistema y (b) explicamos qué hace el sistema a partir de los diferentes componentes funcionales.

A la hora de encontrar los componentes funcionales hay que tener en cuenta que los módulos suelen estar ocupados por componentes funcionales. Además, suele haber componentes funcionales cerca de grandes zonas de comentarios y los identificadores de los componentes funcionales suelen ser largos y formados por palabras entendibles.

Una vez encontrados los posibles componentes funcionales, conviene repasar la lista teniendo en cuenta que un componente es funcional cuando Un componente es funcional cuando su ausencia impide seriamente el funcionamiento de la aplicación, dificulta la legibilidad del código, impide la comprensión de todo o de otro componente funcional o cuando hace caer a niveles muy bajos la calidad, fiabilidad, mantenibilidad, etc.

Vamos a ver cómo a partir de un código java cómo se puede realizar Ingeniería Inversa de Procesos. Tenemos dos clases (Persona y Trabajador)

```
class Persona {

protected String nombre;

protected int edad;

protected int seguroSocial;

protected String licenciaConducir;

public Persona(String nom, int ed, int seg, String
lic) {

set(nom, ed); seguroSocial = seg; licenciaConducir
= lic; }
```

```
public Persona() {
Persona(null, 0, 0, null); }
public int setNombre(String nom) {
nombre = nom; return 1; }
public int setEdad(int ed) {
edad = ed; return 1; }
public void set(String nom, int ed) {
setNombre(nom); setEdad(ed); }
public void set(int ed, String nom) {
setNombre(nom); setEdad(ed); }
}
class Trabajador extends Persona {
private String empresa;
private int salario;
public Trabajador(String emp, int sal) {
empresa = emp; salario = sal; }
public Trabajador() {
this(null,0); }
public int setEmpresa String emp) {
empresa = emp; return 1; }
```

```
public int setSalario(int sal) {
salario = sal; return 1; }
public void set(String emp, int sal) {
setEmpresa(emp); setSalario(sal); }
public void set(int sal, String emp) {
setEmpresa(emp); setSalario(sal); }
}
```

Si realizamos Ingeniería Inversa, el diagrama UML sería el siguiente:

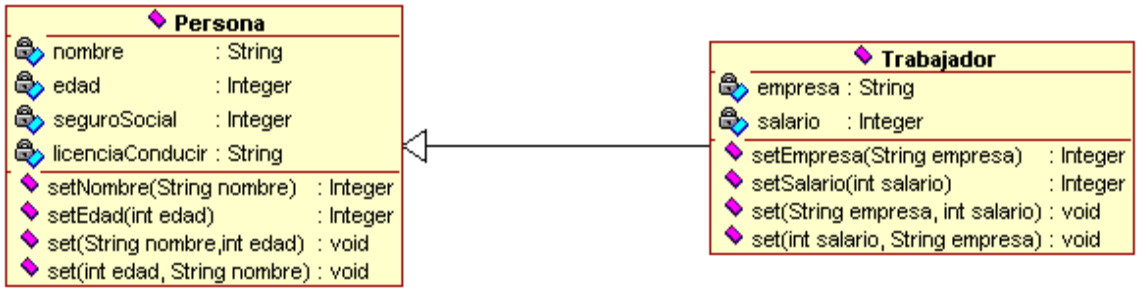


Figura 1. Diagrama de clase generado a partir del código Java

Ingeniería Inversa de Datos

La Ingeniería Inversa de Bases de Datos es el conjunto de técnicas que permite la obtención de una representación conceptual de un esquema de base de datos a partir de su codificación.

Sus aplicaciones son múltiples:

- Re-documentar, reconstruir y/o actualizar documentación perdida o inexistente de bases de datos
- Servir como pivote en un proceso de migración de datos
- Ayudar en la exploración y extracción de datos en bases poco documentadas.

La información que se puede extraer, dependiendo del punto de partida puede ser: Entidad, relaciones, atributos, claves primarias o ajenas, etc., a partir de estos elementos se crean modelos de datos, como por ejemplo Diagramas entidad-relación.

A continuación se muestra un ejemplo gráfico de Ingeniería Inversa de Datos. A partir del código fuente (diseño físico), se realiza Ingeniería Inversa y se obtiene el Diseño Lógico. A este diseño se vuelve a aplicar Ingeniería Inversa y se obtiene el Diseño Conceptual.

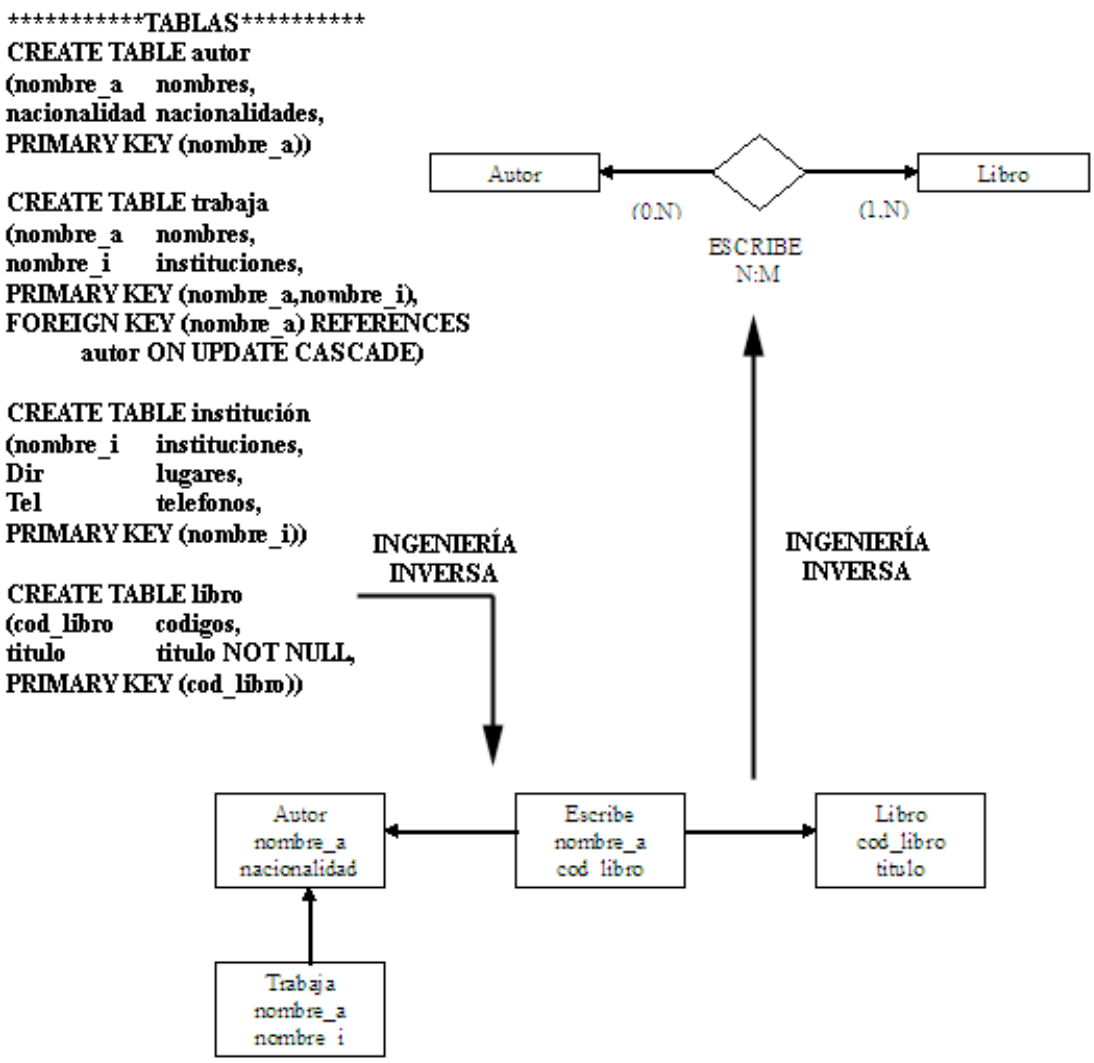


Figura 1. Ejemplo de Ingeniería Inversa de Datos

Técnicas de la Ingeniería Inversa de Datos

La ingeniería inversa de datos se aplica sobre algún código de bases de datos (aplicación, código SQL, etc.) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad-relación. Hay que tener en cuenta que la ingeniería inversa se puede dar entre distintos productos del ciclo de vida de una aplicación.

Existen muchas técnicas para hacer ingeniería inversa de base de datos, algunos de los cuales se pueden ver resumidos en la siguiente tabla:

- Baltín et al (1992):
    - Suposiciones: 3FN. Consistencia en el nombrado de los atributos. Sin homónimos. Clave principal y clave candidata.
    - Entradas: Dependencias. Esquemas de relación.
    - Salidas: Entidades. Relaciones binarias. Categorías. Generalizaciones.
    - Basado en el método de Navathe de Awongs.
- Chiang et al (1994,1996):
    - Suposiciones: 3FN. Consistencia en el nombrado de los atributos. Sin errores en los atributos claves.
    - Entradas: Esquema de relación. Instancia de datos. Dependencias.
    - Salidas: Entidades. Relaciones binarias. Generalizaciones. Agregaciones.
    - Características: Requiere el conocimiento acerca del nombre de los atributos. Propone un marco de trabajo para la evaluación de los métodos de ingeniería inversa. Identifica claramente los casos en los que las entradas de los humanos son necesarias.
- Davids & Arora (1987):
    - Suposiciones: 3FN. Sin homónimos ni sinónimos.
    - Entradas: Esquemas de relación. Restricciones de claves ajenas.
    - Salidas: Conjunto de entidades. Relaciones binarias. Relaciones n-aria.

    - Características: Apunta a una transformación reversible desde el esquema relacional al esquema conceptual.

- Johannessin (1994):

    - Suposiciones: 3FN. Consultas de dominio independientes.
    - Entradas: Dependencias funcionales y de inclusión. Esquemas de relación.
    - Salidas: Generalizaciones. Entidades. Relaciones binarias.
    - Características: Basado en los conceptos establecidos de las teoría de bases de datos relacionales. Proceso de mapeo simple y automático.

- Markowitz & Makowsky (1990):

    - Suposiciones: FN Boycce-Codd.
    - Entradas: Esquemas de relación. Dependencias de claves. Dependencias de integridad referencial.
    - Salidas: Entidades. Relaciones binarias. Generalizaciones y agregaciones.
    - Características: Requiere todas las dependencias de clave.

- Navathe & Among (1987):

    - Suposiciones: 3FN y algunos 2FN. Consistencia en el nombrado de atributos. Sin ambigüedades de clave ajena. Claves candidatas específicas.
    - Entradas: Esquemas de relación.
    - Salidas: Entidades. Relaciones binarias. Categorías. Cardinalidades.
    - Características. Es vulnerable a la ambigüedad en el reconocimiento de claves ajenas.

- Petit et al (1996):

    - Suposiciones: 1FN. Atributos únicos.
    - Entradas: Esquemas de relación. Instancia de datos y código.
    - Salidas: Entidades. Relaciones. Generalizaciones.

    - Características: Analiza las consultas en los programas de aplicación. No pone restricciones en el nombre de los atributos.

- Permerlani & Blaha (1993,1994):

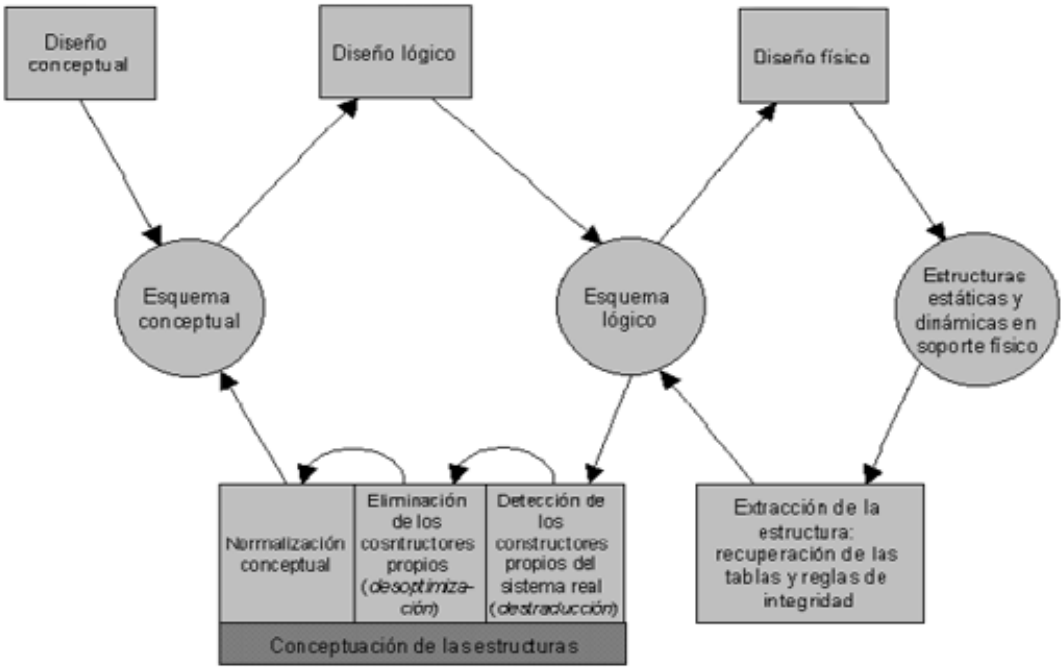
    - Suposiciones: Sin F3N. Comprensión semántica de aplicación.
    - Entradas: Esquemas de relación. Observaciones de patrones de datos.
    - Salidas: Clases. Asociaciones. Generalizaciones. Multiplicidad. Agregación.
    - Características: Requiere un alto nivel de entrada de los humanos. Enfatiza en el análisis de las claves.

- Sotou (1997,1998):

    - Suposiciones: Sin nombres únicos de atributos. Dependencias desconocidas.
    - Entradas: Esquema de datos. Instancia de datos. Diccionario de datos.
    - Salidas: Cardinalidad de las restricciones de relaciones n-arias.
    - Características: Proceso automatizado completamente para SQL.

Entre las distintas técnicas de Ingeniería Inversa de datos, se propone el método de Hainaut et al () para explicarla.

El método pasa por dos fases. En la 1ª fase se realiza la extracción de estructuras y en la 2ª la conceptualización de las mismas:



1. FASE I: Extracción de estructuras

   - Considerar cada fichero una posible tabla.
   - Considerar cada campo del fichero como un posible campo de la tabla.
   - Identificar las claves primarias.
   - Identificar claves ajenas.
   - “Filtrar” las tablas (por ejemplo, despreciar aquellos ficheros sin clave principal).
   - Detección de campos obligatorios.
   - Detección de asociaciones entre tablas

2. FASE II: Conceptuación de las estructuras

   - Sustitución de constructores propios del sistema real por constructores independientes (ej: una tabla que es un elemento físico es sustituida por el concepto de entidad que es un elemento lógico).
   - Detección y eliminación de los constructores no semánticos del esquema lógico, paso inverso a la optimización del esquema (ej: deshacer la normalización de un SGBD relacional).

- Normalización conceptual para obtener estructuras de alto nivel.

Ejemplo de Ingeniería Inversa de Datos

La Ingeniería Inversa de Bases de Datos es el conjunto de técnicas que permite la obtención de una representación conceptual de un esquema de base de datos a partir de su codificación.

Sus aplicaciones son múltiples: Re-documentar, reconstruir y/o actualizar documentación perdida o inexistente de bases de datos, servir como pivote en un proceso de migración de datos, y ayudar en la exploración y extracción de datos en bases poco documentadas.

Ahora se comienza a realizar el análisis por el cual obtendremos el modelo conceptual de una base de datos a partir de un modelo físico.

Esta aplicación está implentada en Delphi, con un Oracle Server 8i Lite, por lo tanto los ejemplos están basados en dichos productos. De todas formas, el análisis es el mismo a seguir independientemente del lenguaje o base de datos que utilicemos.

Lo primero que se debe de hacer es obtener toda la información posible de la estructura de la base de datos (no de los datos que contiene), es decir, nombre de las tablas, atributos de las tablas, etc. Dicha información se encuentra almacenada en el catálogo de la base de datos (el cual se consulta fácilmente utilizando SQL). La información obtenida a partir del catálogo se debe almacenar en algún lado (se suele crear una serie de clases que permitan almacenar toda la información y además a dichas clases se les agrega cierta funcionalidad que permita manejar fácilmente la información almacenada en ellas).

Para realizar la obtención de todas las tablas que componen la base de datos se debe efectuar una consulta SQL. En dicha consulta se obtiene los nombres de las tablas, atributos que componen las tablas con sus características más generales (tipos de datos y si admite valores nulos).

La consulta SQL que utilice es la siguiente:

```
SELECT at.table_name, attc.column_name, attc._data_type, attc.nullable

FROM all_tables at, all_tab_columns attc

WHERE at.table_name = attc.table_name
```

El resultado de dicha consulta será el siguiente: por cada fila habrán cuatro columnas. Las columnas significan lo siguiente: nombre de la tabla, nombre del atributo, tipo de dato del atributo y si el atributo puede ser nulo. Por lo tanto, cada tabla tendrá tantas filas en el resultado de la consulta como atributos posea.

Una vez realizada esta consulta se procede a guardarla en las estructuras de almacenamiento. Una vez hecho se puede analizar cuales atributos de las tablas corresponden a la clave primaria, cuales son claves foráneas y cuales son claves únicas (que en el modelo de normalización serían las claves candidatas).

Para obtener aquellos atributos que componen la clave primaria de una tabla dada se realiza la siguiente consulta, que se debe realizar para cada tabla existente en la base de datos, cambiando en la siguiente consulta NombreTabla por el nombre de la tabla que se consulta (a partir de este momento, cada vez que se coloque NombreTabla se entenderá que es la tabla que nos encontramos analizando). Aquí va la consulta SQL realizada:

```
SELECT column_name

FROM all_constraints ac, all_cons_columns acc

WHERE ac.table_name = 'NombreTabla'

AND ac.constraint_type = 'P'

AND ac.constraint_name = acc.constraint_name
```

```
ORDER BY acc.position;
```

El resultado de esta consulta es una fila para cada atributo que forma parte de la clave primaria. Dichos atributos se desplegarán en orden ascendente según su posición, para así poder ingresarlos en el orden por el cual fueron definidos.

A continuación se obtendrán los atributos que forman las claves foráneas de una tabla, y las tablas a las cuales hace referencia dicha clave foránea perteneciente a una determinada tabla, que se llamará nuevamente NombreTabla.

```
SELECT ac.constraint_name, column_name, r_constraint_name

FROM all_constraints ac, all_cons_columns acc

WHERE ac.table_name = 'NombreTabla'

AND ac.constraint_type = 'R'

AND ac.constraint_name = acc.constraint_name

ORDER BY acc.position;
```

Como resultado se obtiene una fila por cada atributo que compone a una clave foránea. Cada constraint (clave foránea en este caso) tendrá tantas filas como atributos los compongan. Por cada columna tenemos la siguiente información en el siguiente orden: nombre del constraint, nombre del atributo y nombre del constraint al cual hace referencia.

A partir de los constraints a los cuales se hacen referencia, se puede obtener fácilmente a que tabla pertenecen por medio de la siguiente consulta:

```
SELECT table_name

FROM all_constraints

WHERE constraint_name = 'NombreConstraint'
```

Con la consulta anterior obtenemos a que tabla pertenece el constraint NombeConstraint y por lo tanto, si una clave foránea hace referencia al constraint NombreConstraint, entonces ahora sabemos a que tabla hace referencia dicha clave foránea.

Finalmente, para la carga de datos sólo falta averiguar cuales son los atributos que componen a las claves únicas. Para esto se realiza la siguiente consulta:

```
SELECT ac.constraint_name, column_name

FROM all_constraints ac, all_cons_columns acc

WHERE ac.table_name = 'NombreTabla'

AND ac.constraint_type = 'U'

AND ac.constraint_name = acc.constraint_name

ORDER BY acc.position;
```

La consulta anterior devuelve todas las claves únicas que existen en una tabla. Cada clave única tendrá tantas filas en el resultado de la consulta como atributos la compongan. El significado de las columnas es el siguiente: nombre del constraint (o sea, de la clave única en este caso) y nombre del atributo.

Análisis de las tablas

A continuación se presenta como determinar que representación conceptual tiene una tabla dada. Es decir, por ejemplo, una tabla puede ser considerada una entidad, una relación binaria, una relación ternaria, una categorización, etc.

En general, el siguiente análisis debe ser realizado por todas las tablas. Como guía, se presenta un diagrama de flujo, que es el que se debe seguir a la hora de analizar una tabla. Es decir, a una tabla dada se le realizarán ciertas pruebas y en función de los resultados de dichas pruebas decidiremos que 'camino' del diagrama de flujo seguir.

A continuación se presenta el diagrama de árbol que sirve de ayuda a la hora de realizar el análisis.

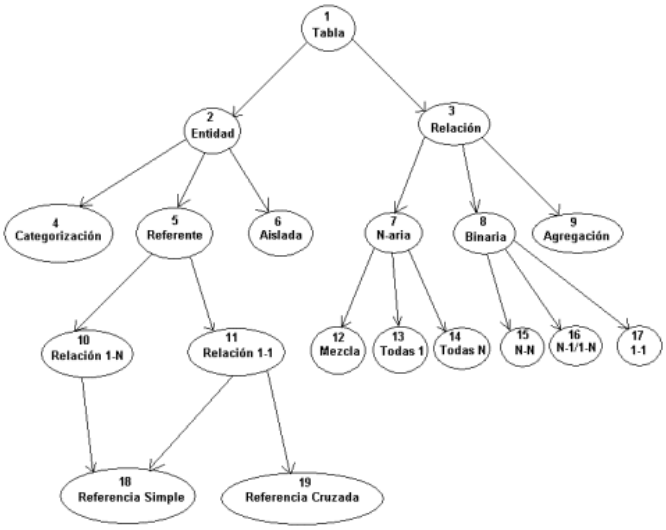


Figura 1. Diagrama de árbol

Determinar si una tabla corresponde a una entidad o a una relación

Lo primero que se debe realizar en el proceso del análisis es determinar si el modelo conceptual de una tabla corresponde a una entidad o a una relación.

Para realizar dicho análisis, se intentan probar distintos casos, mediante los cuales se podrá descartar las diferentes opciones.

Determinar si corresponde a una entidad aislada

Tal vez el término 'aislada' no es el más adecuado, debido a que en un modelo relacional bien hecho, muy difícilmente existan tablas completamente aisladas. En este análisis se refiere a entidades aisladas cuando una tabla no posee claves foráneas a otras tablas. Mediante el análisis de esta tabla no se puede saber a priori las relaciones en las que participa dicha tabla, pero sí se podrá determinar más adelante del análisis. Por lo tanto no es una entidad aislada, sino que más bien es una potencial entidad aislada, pero no se sabrá hasta finalizar el análisis de todas las tablas.

Determinar si una tabla corresponde a un entidad aislada es muy sencillo, lo único que se debe hacer es fijarse si dicha tabla posee claves foráneas. En el caso de que posea estamos seguros de que no es una entidad aislada y podemos proseguir con el análisis de la tabla, pero si se diera el caso que no posee ninguna clave foránea, entonces

estamos seguros que corresponde a una entidad aislada, por lo que podemos agregar dicha tabla a nuestra estructura de almacenamiento entidades y pasar a analizar la siguiente tabla.

Determinar si corresponde a una categorización

Las categorizaciones se caracterizan por lo siguiente: toda la clave primaria de una tabla 'hija' forma una (y solo una) clave foránea a la tabla 'madre'.

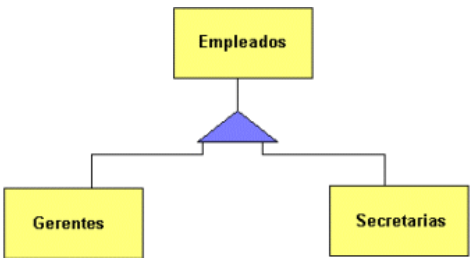
Si se llega a este punto se sabe que la tabla posee por lo menos una clave foránea (por dicha razón no es considerada una entidad aislada). Lo primero que se debe hacer es fijarse si los atributos que componen a la clave primaria de la tabla componen a su vez una clave foránea. Con esto quiero decir que los atributos que componen la clave primaria NO componen a más de una clave foránea. En el caso que los atributos que componen la clave primaria no compongan ninguna clave foránea, o que compongan a más de una clave foránea, se está seguro de que no nos encontramos frente a una categorización.

A continuación se plantea un ejemplo sencillo de categorización mediante el uso de tres tablas: empleados, gerentes y secretarias. La estructura física de las tres tablas es la siguiente:

Empleados	Gerentes	Secretarias
Número_Empleado (PK)	Número_Empleado (PKFK)	Número_Empleado(PKFK)

El atributo Número_Empleado, tanto en la tabla Gerentes como en la tabla Secretarias, forma una clave foránea a la tabla Empleados.

Debido a que tanto la tabla Gerentes como la tabla Secretarias no poseen más claves foráneas, deducimos instantáneamente que no es una tabla que represente una relación, sino que existe una categorización. Por lo tanto, la representación sería la siguiente:



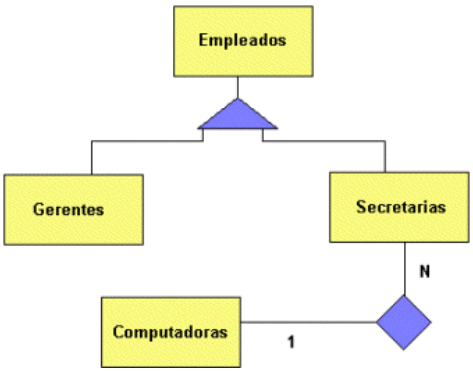
Imaginemos el siguiente caso:

Empleados	Gerentes	Secretarias
Número_Empleado	Número_Empleado	Número_Empleado(PKFK)Número_Computadora(FK)

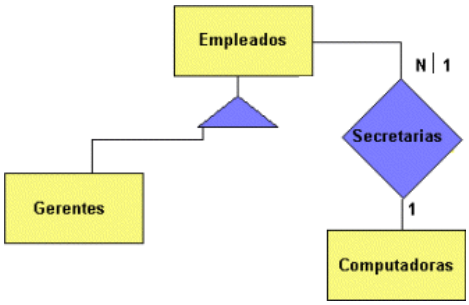
(PK)	(PKFK)

Si se nos diera este caso debemos realizar un análisis más profundo para poder determinar si la tabla Secretarias pertenece a una categorización, debido a que la representación conceptual de esta tabla podría ser cualquiera de las siguientes:

Caso 1:



Caso 2:



Como se puede observar, son representaciones completamente diferentes. En la primera Secretarias forma parte de una categorización, y en la segunda Secretarias es considerada una relación entre Empleados y Computadoras, con cardinalidades N–1 o 1–1.

En ocasiones es posible poder diferenciar entre los dos casos. La única forma de hacerlo es examinando el atributo Número_Computadora que pertenece a la tabla Secretarias y que hace referencia a la tabla Computadoras: si dicho atributo admite valores nulos, estamos completamente seguros que nos encontramos en el primer caso y no en el segundo. Esto es debido a que si Secretarias fuese una tabla que representa una relación entre Empleados y Computadoras, el atributo Número_Computadora NO podría aceptar valores nulos, debido a que por definición, las relaciones binarias son pares ordenados.

Si se nos plantea el caso de que el atributo Número_Computadora perteneciente a la tabla Secretarias no admite valores nulos, es imposible diferenciar entre los dos casos que planteamos anteriormente, por lo tanto debemos adoptar criterios para poder diferenciar entre ellos (por ejemplo, valores por omisión).

Una vez terminada esta parte del análisis sabemos si la tabla pertenece a una categorización o no, si perteneciera a una categorización se almacena en las estructuras de almacenamiento, y se analiza el resto de las claves foráneas que posee la tabla como si se tratase de una tabla que representa a una entidad referente.

Determinar si corresponde a una entidad referente

Entidad referente es una tabla que hace referencia a otras tablas (son las clásicas relaciones 1–N o 1–1).

Si llegamos a este punto sabemos que no nos encontramos frente a una tabla que representa a una entidad aislada y que tampoco corresponde a una categorización. Por lo tanto, ya se está seguro de que esta tabla es una tabla referente dado que tampoco puede representar una relación debido a que en una tabla que represente una relación los atributos que forman la clave primaria de la tabla deben formar también al menos una clave foránea.

Sabemos que cada clave foránea que posea la tabla representará una relación binaria (debido a que es el único tipo de relación que puede representarse sin utilizar una tabla) entre la tabla que nos encontramos analizando y la tabla a la cual hace referencia la clave foránea. Además sabemos que la cardinalidad de dicha relación es 1–1, o bien 1–N, debido a que si fuese N–N se debería haber representado por medio de una tabla. Finalmente también sabemos que la cardinalidad correspondiente a la tabla que nos encontramos analizando es 1.

Análisis de una tabla que representa una relación

El análisis de una relación puede llegar a ser el más complejo debido a la cantidad de casos diferentes que existen (recuerde que una tabla podría representar una relación de N entidades, por lo tanto el número de tablas que podría llegar a relacionar es variable e infinito).

En todos los casos en los cuales una tabla representa una relación nos es imposible determinar las totalidades y parcialidades de la relación. Si se prueba que la tabla es una relación se coloca a esta junto con todos sus datos en nuestras estructuras de almacenamiento de relaciones (por ejemplo, nombre de la relación, tablas que relaciona, cardinalidad, etc.).

Relación binaria 1–1

Para explicar este caso plantearemos la siguiente situación: dada las entidades A y B que se relacionan mediante una relación con cardinalidad 1–1, tenemos que dado un elemento de A sólo existe un elemento de B y dado un elemento de B sólo existe un elemento de A. Ahora, para representar dicha situación mediante una tabla sólo existe una forma, y es la siguiente: una de las foráneas debe ser obligatoriamente la clave primaria (digo una porque recordemos que la clave primaria determina de forma única y mínima a cualquier tupla de la relación, y debido a que queremos representar una relación 1–1), con eso representaríamos una de las cardinalidades 1 (por ejemplo, la de A), pero aún nos falta representar la segunda cardinalidad 1 (siguiendo con el ejemplo la de B). Para realizar esto último debemos hacer uso de las claves candidatas, es decir, debemos hacer que la segunda clave foránea sea a su vez clave única (con esto representaríamos que B también posee clave única).

A continuación se plantea un ejemplo para intentar clarificar este punto.

Vehículos	Matrículas_Vehículos	Matrículas
Número_Vehículo(PK)Número_Matrícula(FK)	Número_Vehículo(PKFK)	Número_Matrícula(PK)

Obviamente la tabla Matrículas_Vehículos intenta representar una relación entre las entidades Vehículos y Matrículas. Como vemos, Número_Vehículo es una clave foránea y a su vez es clave primaria de la tabla, por lo que deducimos que la cardinalidad de Vehículos es 1. Ahora, si queremos representar una relación binaria 1–1 debemos hacer que los atributos que componen a la otra clave foránea (en este caso Número_Matrícula) además de

foráneos sean únicos en la tabla. Con esto último representaríamos que Matrículas también posee cardinalidad 1 en la relación.

A continuación presento un conjunto de tuplas para clarificar la necesidad de poseer la clave única.

Vehículos	Matrículas_Vehículos	Matrículas	Válido
8946	Num_Veh: 8946Num_Mat: SAB 555	SAB 555	Sí
8946	Num_Veh: 8946Num_Mat: SAK 430	SAK 430	No
1388	Num_Veh: 1388Num_Mat: SAK 430	SAK 430	No

Como se ve en el ejemplo de tuplas anterior, existe una necesidad de especificar el atributo Número_Matrícula como único.

En teoría deberíamos especificar las dos claves foráneas como únicas, pero debido a que la definición de clave primaria es que es única y no nula, queda implícito que si una clave foránea debe ser a su vez clave primaria, entonces dicha clave foránea también es única.

Relación binaria N–1 / 1–N

En este tipo de relación solo poseemos dos claves foráneas. Para esta situación, los atributos que componen la clave foránea correspondiente a la entidad que posee cardinalidad N deben formar a su vez la clave primaria de la tabla. A diferencia de el caso anterior, los atributos que forman la clave foránea correspondiente a la otra entidad NO pueden ser declarados como únicos.

Relación binaria N–N

A diferencia de los casos anteriores, este tipo de relación sí puede formar agregaciones, y debemos hacer ciertas consideraciones antes de comenzar su análisis.

Para representar este tipo de relación siempre se debe utilizar una tabla, y los atributos que compongan las claves foráneas correspondientes a las dos tablas que relaciona deben formar a su vez la clave primaria de la tabla.

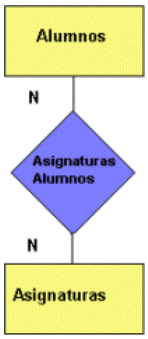
En el caso que la clave primaria este formada por una sola clave foránea, y que a su vez no todos los atributos de dicha clave foránea formen a la clave primaria, podemos considerar que se quiere representar a una entidad no representada (es decir, que dicha entidad existe en el modelo conceptual, pero no en el físico y lógico).

A continuación se presenta un caso sencillo de este tipo de relación.

Alumnos	Asignaturas_Alumnos	Asignaturas
Número_Alumno(PK)Número_Asignatura(PKFK)	Número_Alumno(PKFK)	Número_Asignatura(PK)

Se observa que la clave primaria de la tabla Asignaturas_Alumnos se encuentra formada por dos claves foráneas. Debido a que la tabla Asignaturas_Alumnos no posee ninguna clave foránea a excepción de las que componen a la clave primaria, deducimos rápidamente que nos encontramos frente a una relación binaria N–N.

Por lo tanto, la representación conceptual de las tablas anteriores es la siguiente:



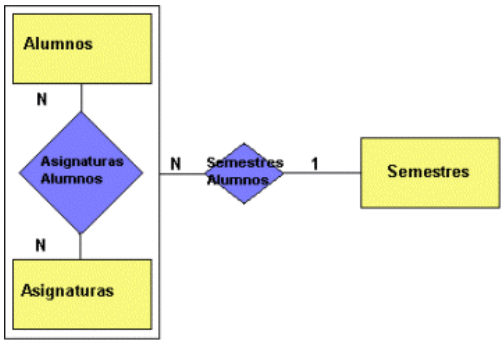
El caso de ejemplo que planteamos anteriormente ilustra un caso típico, en el cual podemos deducir sin ambigüedades la cardinalidad y tipo de relación existente, pero imagínese el siguiente caso:

Alumnos	Asignaturas_Alumnos
Número_Alumno(PK)	Número_Alumno (PKFK)Número_Asignatura (PKFK) Número_Semestre (FK)

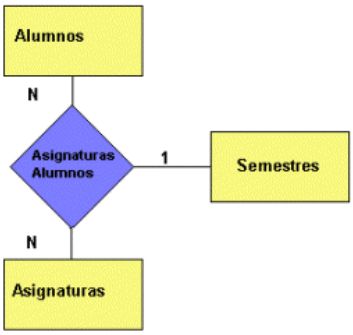
Asignaturas	Semestres
Número_Asignatura (PK)	Número_Semestre (PK)

La tabla Asignaturas_Alumnos posee las siguientes características: la clave primaria de la tabla se compone por dos claves foráneas, pero además posee una clave foránea que no compone a la primaria. Es obvio que nos encontramos frente a una tabla que representa una relación, pero el problema es determinar el tipo de relación existente. Si nos enfrentamos a un caso similar a este, se nos pueden plantear dos posibilidades que mostramos a continuación:

Caso 1:



Caso 2:



Como notará, existe una gran diferencia entre las dos posibilidades, debido a que la primera corresponde a una relación binaria formando una agregación, pero en el segundo caso la relación es ternaria. Está situación se plantea debido a que si una relación forma una agregación que se relaciona con otra entidad, si dicha relación (entre la agregación y la entidad) posee cardinalidades 1–1 o N–1, existe la posibilidad de que no se represente la relación por medio de una tabla.

Para poder resolver este problema sólo tenemos dos posibilidades. La primera es examinando los atributos que forman la clave foránea que no compone la clave primaria de la tabla que representa la relación y en el caso de que dichos atributos admitan valores nulos, entonces deducimos directamente que nos encontramos frente al caso de una agregación, debido a que si fuese una relación ternaria no debería admitir valores nulos. Si la primer opción falla, tenemos una ultima opción y es intentar probar una relación 1–1 cruzada entre la relación y la entidad (en nuestro ejemplo entre la tabla Asignaturas_Alumnos y Semestres) en el caso de que probemos que existe una relación 1–1 cruzada podemos decir con total seguridad que nos encontramos frente a una agregación.

Análisis de una relación n–aria

Si llegamos a este punto, sabemos con certeza de que la tabla representa una relación, y que a su vez esta relación no es binaria. Por ende, nos queda sólo determinar si es o no una agregación, y en el caso que no lo sea analizar las características de la relación (cardinalidad, entidades que relaciona, etc.).

Para realizar este análisis consideraremos que una relación n–aria es una relación que relaciona N entidades, siendo N un número mayor que dos (esto lo haremos porque las binarias las analizamos en la sección anterior, pero no se debe perder de vista que una relación binaria es un caso particular de una relación n–aria). A pesar de este hecho, existen tres posibilidades bien diferenciadas en una relación n–aria, vinculadas con sus cardinalidades: la primera es que todas sus cardinalidades sean uno, la segunda es que todas sus cardinalidades sean N y finalmente que sus cardinalidades sean una mezcla entre unos y enes.

Relación n–aria con todas sus cardinalidades N

Si una tabla representa una relación n–aria con todas sus cardinalidades N, entonces sabemos con certeza de que dicha tabla no posee claves únicas para representar su cardinalidad (debido a que no necesita esto).

Distinguir este tipo de relación es sumamente fácil, debido a que para representar esta cardinalidad todas las claves foráneas que forman la relación deben formar a su vez la clave primaria de la tabla.

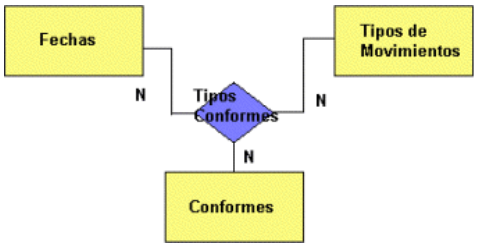
Ahora puede darse el caso de que se quiera representar una relación entre varias entidades, en donde una de estas entidades no se represente tanto en el modelo lógico como en el físico. No tenemos duda que es una relación n–aria con todas sus cardinalidades N, debido a que no posee claves únicas y todas las claves foráneas forman la clave primaria. Pero no todos los atributos que componen a la clave primaria son foráneos, por lo tanto deducimos que existen entidades no representadas en el modelo físico. Aquí se evaluarán dichos atributos según los criterios que nosotros impongamos (por ejemplo, cada atributo es una entidad no representada, preguntar al usuario, etc.).

A continuación planteo un ejemplo, de una relación n–aria con entidades no representadas:

Tipos_de_Movimientos	Conformes	Tipos_Conformes
Número_Tipo (PK)	Número_Conforme (PK)	Número_Tipo (PKFK)Número_Conforme (PKFK)Fecha (PK)

En las tablas anteriores, notamos que en la tabla Tipos_Conformes el atributo Fecha forma parte de la clave primaria de la tabla, pero no compone ninguna clave foránea, por lo tanto deducimos que es una entidad no representada (no sería lógico tener en el modelo físico una tabla con todas las fechas posibles; en cambio, en el modelo conceptual sí es útil y muchas veces necesario).

La representación conceptual de la tabla anteriormente expuesta es la siguiente:



Relación n–aria con todas sus cardinalidades 1

Para determinar si una relación pertenece a este tipo debemos estudiar sus claves candidatas (es decir, sus claves únicas). En este caso sabemos que toda clave foránea que pertenezca a una entidad que esta tabla relaciona se debe encontrar ya sea en la clave primaria o en alguna de sus claves únicas.

A continuación planteamos un ejemplo de este tipo de relación por medio de una relación ternaria:

Matrículas	Vehículos
Número_Matrícula (PK)	Número_Matrícula (PK)

Departamentos	Matrículas_Vehículos_Departamentos
Número_Departamento (PK)	Número_Matrícula (PKFK)Número_Vehículo (PKFK)Número_Departamento (FK)

En este caso existirán dos claves únicas además de la clave primaria. Dichas claves únicas son (Número_Matrícula, Número_Departamento) y (Número_Vehículo, Número_Departamento).

Entonces en este caso deducimos que la relación es 1-1-1 debido a que al haber una clave foránea que no es primaria, entonces sabemos que la cardinalidad de la entidad a la cual hace referencia dicha clave foránea es 1. Ahora listamos todas las claves únicas con los atributos que la componen. Para cada caso aquel atributo que no se encuentre en una clave única y que nosotros sepamos que hace referencia a una entidad que relaciona la tabla, entonces sabemos con certeza que tiene cardinalidad 1, es decir, si Número_Matrícula-Número_Departamento componen una clave única sabemos que Número_Vehículo tiene cardinalidad 1.

Ahora pasaremos a justificar lo anterior con un ejemplo, ingresando algunas tuplas a las tablas de la relación antes citada.

Matrículas_Vehículos_Departamentos			Válido
Núm_Mat	Núm_Veh	Núm_Dep	
SAK 445	4670	10	Sí
SAK 445	890	10	No
SSD 320	430	11	Sí
DFF 440	4670	10	No

Debido a que Número_Matrícula, Número_Vehículo son clave primaria de la tabla, entonces deducimos que la entidad Departamentos tiene cardinalidad 1.

Ahora, a partir del ejemplo que mostramos ingresando tuplas, deducimos que un valor de Número_Vehículo y Número_Departamento estos sólo pueden aparecer juntos en una misma tupla una sola vez en toda la tabla. Por ende estos dos atributos forman una clave única, deduciendo entonces que la entidad Matrículas tiene cardinalidad 1. De la misma forma ocurre con la entidad Vehículos.

Por lo tanto la representación conceptual de este conjunto de tablas es la siguiente:

Relación n–aria con sus cardinalidades mezcladas

Se entiende por cardinalidad mezclada la cardinalidad de la relación no es ni todas uno ni todas enes, sino que la cantidad de cardinalidades unos y enes son variables.

Para resolver este tipo de relación, nuevamente haremos uso de las claves candidatas (claves únicas). El análisis lo haré basándome en un ejemplo.

Personas	Personas_Garantes
Número_Persona (PK)	Número_Persona_Garante (PK)

Conformes	Conformes_Personas
Número_Conforme (PK)	Número_Persona (PKFK)Número_Conforme (PKFK)Número_Persona_Garante (FK)

Como podemos observar en los esquemas que describimos anteriormente, la tabla que representa la relación tiene la siguiente clave primaria compuesta: Número_Persona, Número_Conforme; por lo cual deducimos directamente que la cardinalidad de la entidad Personas_Garantes es 1.

Luego tras analizar las claves únicas que posee la tabla deducimos que posee una clave única compuesta por los siguientes atributos: Número_Conforme, Número_Persona_Garante, por lo cual deducimos que la entidad Personas también posee cardinalidad 1. Finalmente al no poseer más claves únicas llegamos a la conclusión de que la cardinalidad de esta relación es 1-1-N.

Conclusión del ejemplo

Se ha podido observar todo el análisis exhaustivo que se ha realizado a través del código fuente (diseño físico), se puede obtener el diseño lógico aplicando Ingeniería Inversa y también el Diseño Conceptual.

Ingeniería Inversa de Interfaces de Usuario

Muchos programas gozan de gran fiabilidad, capacidad de procesamiento, etc., pero sus diseñadores han olvidado la comodidad y facilidad de uso del usuario final (usabilidad). En dichos casos es posible aplicar ingeniería inversa sobre la interfaz de usuario con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.

En general, como resultado de este tipo de procesos se obtiene la relación entre los distintos componentes de la interfaz de usuario, siendo interesante poder obtener aspectos específicos de modelo de interfaces por separado: modelo de tareas, modelo de presentación,… No obstante, la consecución de este objetivo depende en gran medida de la especificación que se utilizara para generar la interfaz en su momento.

Es importante indicar que una interfaz grafica de usuario de sustitución puede que no refleje la interfaz antigua de forma exacta (de hecho, puede ser totalmente diferente). Con frecuencia, merece la pena desarrollar metáforas de interacción nuevas. Por ejemplo, una solicitud de interfaz de usuario antigua en la que un usuario proporcione un factor superior (del 1 al 10) para encoger o agrandar una imagen gráfica. Es posible que una interfaz grafica de usuario diseñada utilice una barra de imágenes y un ratón para realizar la misma función.

¿Qué es Reingeniería del Software?

Reingeniería del software se puede definir como: “modificación de un producto software, o de ciertos componentes, usando para el análisis del sistema existente técnicas de Ingeniería Inversa y, para la etapa de reconstrucción, herramientas de Ingeniería Directa, de tal manera que se oriente este cambio hacia mayores niveles de facilidad en cuanto a mantenimiento, reutilización, comprensión o evaluación.”

Cuando una aplicación lleva siendo usada años, es fácil que esta aplicación se vuelva inestable como fruto de las múltiples correcciones, adaptaciones o mejoras que han podido surgir a lo largo del tiempo. Esto deriva en que cada vez que se pretende realizar un cambio se producen efectos colaterales inesperados y hasta de gravedad, por lo que se hace necesario, si se prevé que la aplicación seguirá siendo de utilidad, aplicar reingeniería a la misma.

Entre los beneficios de aplicar reingeniería a un producto existente se puede incluir:

- Pueden reducir los riegos evolutivos de una organización.
- Puede ayudar a las organizaciones a recuperar sus inversiones en software.
- Puede hacer el software más fácilmente modificable
- Amplía las capacidades de las herramientas CASE
- Es un catalizador para la automatización del mantenimiento del software
- Puede actuar como catalizador para la aplicación de técnicas de inteligencia artificial para resolver problemas de reingeniería

La reingeniería del software involucra diferentes actividades como son:

- análisis de inventarios
- reestructuración de documentos
- ingeniería inversa
- reestructuración de programas y datos
- ingeniería directa

con la finalidad de crear versiones de programas ya existentes que sean de mejor calidad y los mismos tengan una mayor facilidad de mantenimiento.

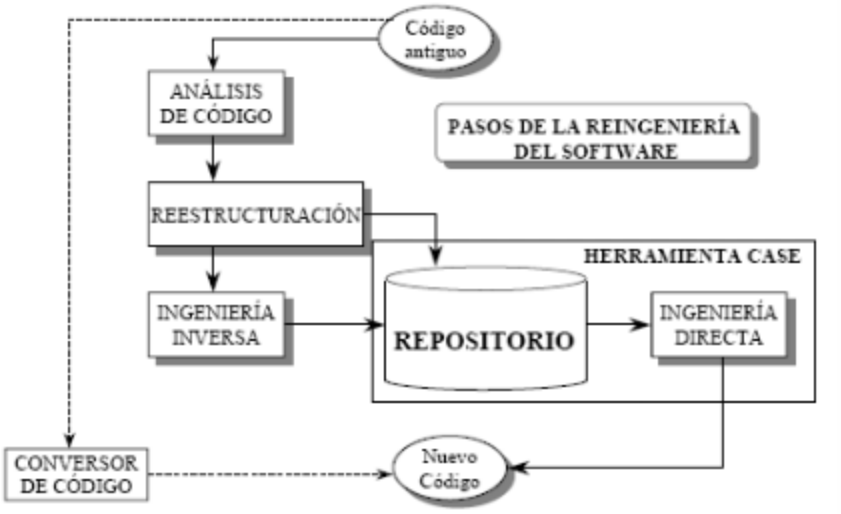


Figura 1. Pasos de la Reingeniería del Software

Análisis de Inventarios

Todas las organizaciones de software deberían tener un inventario de todas sus aplicaciones. El inventario tal vez no sea más que un modelo en una hoja de cálculo que contenga información que proporcione una descripción detallada (tamaño, edad, importancia para el negocio) de las aplicaciones activas.

Los candidatos a la reingeniería aparecen cuando se ordena esta información en función de su importancia para el negocio, longevidad, mantenibilidad actual y otros criterios localmente importantes. Es entonces cuando es posible asignar recursos a las aplicaciones candidatas para el trabajo de reingeniería.

Es importante señalar que el inventario deberá visitarse con regularidad, el estado de las aplicaciones puede cambiar en función del tiempo y, como resultado, cambiarán las prioridades para la reingeniería.

Reestructuración de documentos

La documentación débil es la marca de muchos sistemas heredados. ¿Pero que se hace acerca de ellos? ¿Cuáles son las opciones? Crear documentación consume mucho tiempo, si el sistema funciona vivirá con lo que tenga. La documentación debe actualizarse pero se tiene recursos limitados. Se utiliza un enfoque de “documentar cuando se toque”. El sistema es crucial para el negocio y debe volver a documentarse por completo incluso en este caso un enfoque inteligente es recortar la documentación a un mínimo esencial. Cada una de estas opciones es viable. Una organización de software debe elegir la más apropiada para cada caso.

Ingeniería Inversa

Este término tiene sus orígenes en el mundo del hardware. Una cierta compañía desensambla un producto de hardware competitivo en un esfuerzo por comprender los “secretos” del diseño y fabricación de su competidor. Estos secretos se podrán comprender más fácilmente si se obtuvieran las especificaciones de diseño y fabricación del mismo. Pero estos documentos son privados, y no están disponibles para la compañía que efectúa la ingeniería inversa. En esencia, una ingeniería inversa con éxito precede de una o más especificaciones de diseño y fabricación para el producto, mediante el examen de ejemplos reales de ese producto.

La ingeniería inversa del software es algo similar. En la mayoría de los casos, el programa del cual hay que hacer una ingeniería inversa no es el de un rival, sino, más bien, el propio trabajo de la compañía. Los “secretos” que hay que comprender resultan incomprensibles porque nunca se llegó a desarrollar una especificación. Consiguientemente, la ingeniería inversa del software es el proceso de análisis de un programa con el fin de crear una representación de programa con un nivel de abstracción más elevado que el código fuente.

La Ingeniería inversa es un proceso de recuperación de diseño. Con las herramientas de la ingeniería inversa se extraerá del programa existente información del diseño arquitectónico y de proceso, e información de los datos.

Reestructuración de código

El tipo más común de reingeniería es la reestructuración de código, se puede hacer con módulos individuales que se codifican de una manera que dificultan comprenderlos, probarlos y mantenerlos.

Llevar a cabo esta actividad requiere analizar el código fuente empleando una herramienta de reestructuración, se indican las violaciones de las estructuras de programación estructurada, y entonces se reestructura el código (esto se puede hacer automáticamente). El código reestructurado resultante se revisa y se comprueba para asegurar que no se hayan introducido anomalías. Se actualiza la documentación interna del código.

Reestructuración de datos

La reestructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de datos comienza con una actividad de ingeniería inversa. La arquitectura de datos actual se analiza con minuciosidad y se define los modelos de datos necesarios, se identifican los objetivos de datos y los atributos, y después se revisa la calidad de las estructuras de datos existentes.

Cuando la estructura de datos es débil (por ejemplo, actualmente se implementan archivos planos, cuando un enfoque relacional simplificaría muchísimo el procesamiento), se aplica una reingeniería a los datos.

Dado que la arquitectura de datos tiene una gran influencia sobre la arquitectura del programa, y también sobre los algoritmos que lo pueblan, los cambios en datos darán lugar invariablemente a cambios o bien de arquitectura o bien de código.

Ingeniería directa

En un mundo ideal, las aplicaciones se reconstruyen utilizando un “motor de reingeniería” automatizado. En el motor se insertaría el programa viejo, que lo analizaría, reestructuraría y después regeneraría la forma de exhibir los mejores aspectos de la calidad del software. Después de un espacio de tiempo corto, es probable que llegue a aparecer este “motor”, pero los fabricantes de CASE han presentado herramientas que proporcionan un

subconjunto limitado de estas capacidades y que se enfrentan con dominios de aplicaciones específicos. Lo que es más importante, estas herramientas de reingeniería cada vez son más sofisticadas.

La ingeniería directa no solo recupera la información de diseño a partir del software existente, también utiliza esta información para alterar o reconstruir el sistema existente con la finalidad de mejorar su calidad global. En la mayoría de los casos el software sometido a reingeniería vuelve a implementar la función del sistema existente y también añade nuevas funciones o mejoras.

Procesos involucrados en la Reingeniería del Software

La reingeniería debe ser entendida como un proceso mediante el cual se mejora un software existente haciendo uso de técnicas de ingeniería inversa y reestructuración de código.

Para llevar a cabo la reingeniería del Software se puede realizar a través del modelo Cíclico. Este modelo define seis actividades las cuales se muestran en la figura de abajo. En algunas ocasiones, estas actividades se producen de forma secuencial y lineal, pero esto no siempre es así. Por ejemplo, puede ser que la ingeniería inversa (la comprensión del funcionamiento interno de un programa) tenga que producirse antes de que pueda comenzar la reestructuración de documentos.

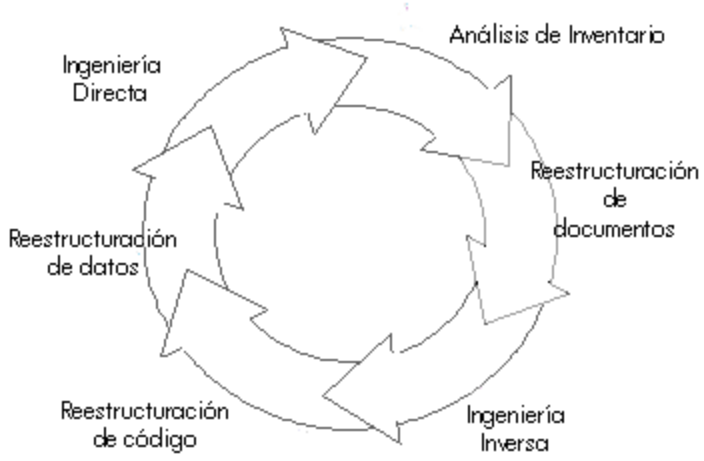


Figura 1. Modelo Cíclico de la Reingeniería del Software

El paradigma de la reingeniería mostrado en la figura es un modelo cíclico. Esto significa que cada una de las actividades presentadas como parte del paradigma pueden repetirse en otras ocasiones. Para un ciclo en particular, el proceso puede terminar después de cualquier de estas actividades.

Análisis de inventario

Todas las organizaciones de software deberán disponer de un inventario de todas sus aplicaciones. El inventario puede que no sea más que una hoja de calculo con la información que proporciona una descripción detallada (por

ejemplo: tamaño, edad, importancia para el negocio) de todas las aplicaciones activas.

Los candidatos a la reingeniería aparecen cuando se ordena esta información en función de su importancia para el negocio, longevidad, mantenibilidad actual y otros criterios localmente importantes. Es entonces cuando es posible asignar recursos a las aplicaciones candidatas para el trabajo de reingeniería.

Es importante destacar que el inventario deberá revisarse con regularidad. El estado de las aplicaciones (por ejemplo, la importancia con respecto al negocio) puede cambiar en función del tiempo y, como resultado, cambiarán también las prioridades para la reingeniería.

Reestructuración de documentos

Una documentación escasa es la marca de muchos sistemas de información heredados. ¿Qué se puede hacer al respecto?

- Opción 1: La creación de documentación consume muchísimo tiempo. El sistema funciona, y ya nos ajustaremos con lo que se tiene. En algunos casos, éste es el enfoque correcto. No es posible volver a crear la documentación para cientos de programas de computadoras. Si un programa es relativamente estático está llegando al final de vida útil, y no es probable que experimente muchos cambios.
- Opción 2: Es preciso actualizar la documentación, pero se dispone de recursos limitados. Se utilizará un enfoque “del tipo documentar si se modifica”. Quizá no es necesario volver a documentar por completo la aplicación. Más bien se documentarán por completo aquellas partes del sistema que estén experimentando cambios en ese momento. La colección de documentos útil y relevante irá evolucionando con el tiempo.
- Opción 3: El sistema es fundamental para el negocio, y es preciso volver a documentarlo por completo. En este caso, un enfoque inteligente consiste en reducir la documentación al mínimo necesario.

Todas y cada una de estas opciones son viables. Las organizaciones del software deberán seleccionar aquella que resulte más adecuada para cada caso.

Ingeniería inversa

El término “ingeniería inversa” tiene sus orígines en el mundo del hardware. Una cierta compañía desensambla un producto de hardware competitivo en un esfuerzo por comprender los “secretos” del diseño y fabricación de su competidor. Estos secretos se podrán comprender más fácilmente si se obtuvieran las especificaciones de diseño y fabricación del mismo. Pero estos documentos son privados, y no están disponibles para la compañía que efectúa la ingeniería inversa. En esencia, una ingeniería inversa con éxito precede de una o más especificaciones de diseño y fabricación para el producto, mediante el examen de ejemplos reales de ese producto.

La ingeniería inversa del software es algo bastante similar. Sin embargo, en la mayoría de los casos, el programa del cual hay que hacer una ingeniería inversa no es el de un rival, sino, más bien, el propio trabajo de la compañía (con frecuencia efectuado hace muchos años). Los “secretos” que hay que comprender resultan incomprensibles porque nunca se llegó a desarrollar una especificación. Consiguientemente, la ingeniería inversa del software es el proceso de análisis de un programa con el fin de crear una representación de programa con un nivel de abstracción más elevado que el código fuente. La ingeniería inversa se extraerá del programa existente información del diseño arquitectónico y de proceso, e información de los datos.

Reestructuración del código

El tipo más común de reingeniería es la reestructuración del código. Algunos sistemas heredados tienen una arquitectura de programa relativamente sólida, pero los módulos individuales han sido codificados de una forma que hace difícil comprenderlos, comprobarlos y mantenerlos. En estos casos, se puede reestructurar el código ubicado dentro de los módulos sospechosos.

Para llevar a cabo esta actividad, se analiza el código fuente mediante una herramienta de reestructuración, se indican las violaciones de las estructuras de programación estructurada, y entonces se reestructura el código (esto se puede hacer automáticamente). El código reestructurado resultante se revisa y se comprueba para asegurar que no se hayan introducido anomalías. Se actualiza la documentación interna del código.

Reestructuración de datos

Un programa que posea una estructura de datos débil será difícil de adaptar y de mejorar. De hecho, para muchas aplicaciones, la arquitectura de datos tiene más que ver con la viabilidad a largo plazo del programa que el propio código fuente.

A diferencia de la reestructuración de código, que se produce en un nivel relativamente bajo de abstracción, la estructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de datos comienza por una actividad de ingeniería inversa. La arquitectura de datos actual se analiza minuciosamente y se definen los modelos de datos necesarios. Se identifican los objetos de datos y atributos y, a continuación, se revisan las estructuras de datos a efectos de calidad.

Cuando la estructura de datos es débil (por ejemplo, actualmente se implementan archivos planos, cuando un enfoque relacional simplificaría muchísimo el procesamiento), se aplica una reingeniería a los datos.

Dado que la arquitectura de datos tiene una gran influencia sobre la arquitectura del programa, y también sobre los algoritmos que los pueblan, los cambios en datos darán lugar invariablemente a cambios o bien de arquitectura o bien de código.

Ingeniería directa

En un mundo ideal, las aplicaciones se reconstruyen utilizando un “motor de reingeniería” automatizado. En el motor se insertaría el programa viejo, que lo analizaría, reestructuraría y después regeneraría la forma de exhibir los mejores aspectos de la calidad del software. Después de un espacio de

tiempo corto, es probable que llegue a aparecer este “motor”, pero los fabricantes de CASE han presentado herramientas que proporcionan un subconjunto limitado de estas capacidades y que se enfrentan con dominios de aplicaciones específicos (por ejemplo, aplicaciones que han sido implementadas empleando un sistema de bases de datos específico). Lo que es más importante, estas herramientas de reingeniería cada vez son más sofisticadas.

La ingeniería directa, que se denomina también renovación o reclamación, no solamente recupera la información de diseño de un software ya existente, sino que, además, utiliza esta información en un esfuerzo por mejorar su calidad global. En la mayoría de los casos, el software procedente de una reingeniería vuelve a implementar la funcionalidad del sistema existente, y añade además nuevas funciones y/o mejora el rendimiento global.

Valoraciones sobre la Reingeniería del Software

Se entiende por reingeniería la modificación de un producto software o de ciertos componentes, usando para el análisis del sistema existente técnicas de ingeniería inversa y para la etapa de reconstrucción, herramientas de ingeniería directa.

La reingeniería requiere tiempo; conlleva un coste de dinero enorme y absorbe recursos que de otro modo podrían emplearse en preocupaciones más inmediatas. La reingeniería es una actividad que absorberá recursos de las tecnologías de la información durante un período de tiempo grande.

Ante la perspectiva de aplicar procesos de reingeniería, cabe preguntarse si existen alternativas a esto:

- Comprar un software que lo sustituya.
- Desarrollar el software de nuevo.

Desde luego, cualquier opción (incluyendo la reingeniería) incurre en costes de mantenimiento y de operaciones. No obstante, la reingeniería se suele considerar una buena opción frente al desarrollo de una nueva aplicación cuando:

- La aplicación tiene fallos frecuentes que son difíciles de localizar.
- La aplicación es poco eficiente, pero realiza la acción esperada.
- Existen dificultades para integrar la aplicación con otros sistemas.
- El software final de la aplicación es de poca calidad.
- Cuando no se dispone de personal suficiente para realizar todas las modificaciones necesarias que puedan surgir.
- Cuando no se tenga facilidad para realizar las pruebas a los cambios que se deban realizar.
- Cuando el mantenimiento de la aplicación consume muchos recursos.
- Cuando es necesario incluir nuevos requisitos a la aplicación, pero los fundamentales se mantienen.

Método cuantitativo

De manera objetiva, se puede calcular el beneficio cuantitativo tanto para comprar un software que lo sustituya como para el desarrollo del software nuevo.

Si se mantiene el software como está, el beneficio se puede calcular de la siguiente manera:

BM = [VA – (CMA + COpA)] x T. Vida

Siendo

BM: el beneficio de mantenimiento

VA: el valor de negocio actual (anual)

CMA: el coste de mantenimiento actual

COpA: el coste actual de operación de la aplicación, es decir, los costes derivados de mantener la aplicación en uso (servicios de atención al cliente, administración,...).

Si por el contrario se elige hacer reingeniería, el beneficio obtenido será:

BR = [(GF x T. Vida) – (CR x FR)] – BM

GF = VF – (CMF x COpF)] – BM

T. Vida = T. Vida Estimado –T. Reingeniería

Donde:

BR: beneficio de reingeniería

GF: ganancia final

CR: coste de reingeniería

FR: factor de riesgo de la reingeniería

BM: beneficio de mantenimiento

VF: el valor de negocio tras la reingeniería (anual)

CMF: coste de mantenimiento final

COpF: coste de operación final

Importancia de aplicar Reingeniería del Software

Mucha gente al ver las grandes y viejas mansiones queda asombrado de su belleza, pero no se preguntan que tan bien se puede vivir en ellas. Las personas que lo hacen dicen que es una pesadilla mantenerlas. Todas ellas fueron construidas con viejas tecnología estándar. Sus paredes externas no tienen aislamiento. El alambrado eléctrico tiene limitaciones y claramente es inadecuada para las necesidades de energía de hoy y su cableado decadente crea un severo peligro eléctrico.

Los viejos sistemas son muy similares a los grandes y viejos edificios. Ellos tienen los mismos problemas de mantenimiento, un hecho en gran parte irreconocible por parte de la comunidad corporativa. Muchos de esos edificios son demolidos por que no son mantenibles y ya no sirven para las necesidades de sus ocupantes.

Las viejas computadoras tal vez se puedan ver solamente en museos. Pero en muchos casos, software escrito para viejos modelos de computadora están ejecutándose hoy en día. Un caso extremo es el de un software escrito para una IBM 1401 Autocoder. Cuando la compañía remplazó la 1401 con una IBM 360/40, compraron un emulador de la 1401 para poder ejecutar el software. Esa aplicación hoy día corre en una PC – la compañía compro otro emulador.

Los clientes demandan que las nuevas capacidades sean agregadas al código escrito en sus viejos sistemas. Casi siempre, las empresas encuentran que no pueden modificar su código – el programador que lo mantenía murió recientemente o nadie sabe programar en el lenguaje en el que fue escrito. Por lo que la funcionalidad de ese programa quedará así para siempre.

La siguiente lista son las razones por las que es aplicable la reingeniería a los sistemas de información heredados:

- Frecuentes fallas de producción (fiabilidad cuestionable).
- Problemas de rendimiento.
- Tecnología obsoleta.
- Problemas de integración del sistema.
- Código de calida pobre.
- Dificultad (peligroso) al cambio.
- Dificultad para probar.
- Mantenimiento caro.
- Incremento de problemas del sistema.

Estas razones pueden ser solucionadas al aplicar un proceso de mantenimiento de software, pero cuando dicho mantenimiento deja de ser viable, entonces se toma la decisión de aplicar reingeniería.

Aunque la reingeniería se usa principalmente durante el mantenimiento del software, va mas allá de una simple ayuda para el mantenimiento. La reingeniería es el puente desde viejas tecnologías hacía nuevas tecnologías que las organizaciones deben usar en la actualidad para responder al cambio de requerimientos del negocio.

Los viejos programas representan la tecnología del ayer. Ahora sabemos que los años tienen cuatro dígitos y no dos, que los datos pueden ser manejados mejor en bases de datos y que tenemos nuevos diseños de construcción y lenguajes de programación que permiten diseñar programas notablemente mantenibles.

Cuando el costo de mantener viejos edificios es altamente excesivo, se remplazan estos edificios. Nosotros deberíamos hacer lo mismo con los programas. Los programas no se hacen obsoletos al paso del tiempo ya que fueron escritos para hardware y sistemas operativos que ya no existen, muchos están llenos de características y parches no documentados. Sólo cuando hayamos aprendido a que es mejor invertir en nuevo hardware y nuevos edificios podremos reconocer el valor de remplazar los viejos sistemas raquíticos.

¿Qué es Reestructuración del Software?

La reestructuración del software modifica el código fuente y/o los datos en un intento de adecuarlo a futuros cambios. Tiende a centrarse en los detalles de diseño de módulos individuales y en estructuras de datos locales definidas dentro de los módulos.

Los beneficios de de la reestructuración son:

- Programas de mayor calidad con mejor documentación y menos complejidad, y ajustados a las prácticas y estándares de la ingeniería del software moderno.
- Reduce la frustración entre ingenieros del software que deban trabajar con el programa, mejorando por tanto la productividad y haciendo más sencillo el aprendizaje.
- Reduce el esfuerzo requerido para llevar a cabo las actividades de mantenimiento.
- Hace que el software se mas sencillo de comprobar y depurar.

La reestructuración se produce cuando la arquitectura básica de la aplicación es sólida, aún cuando sus interioridades técnicas necesiten un retoque. Comienza cuando existen partes considerables del software que son útiles todavía y solamente existe un subconjunto de todos los módulos y datos que requieren una extensa modificación.

Los tipos de reestructuración, básicamente son 2: del código y de datos.

Reestructuración del código

La reestructuración del código se lleva a cabo para conseguir un diseño que produzca la misma función pero con mayor calidad que el programa original.

El objetivo es tomar el código de forma de "plato de espaguetis" y derivar un diseño de procedimientos que se ajuste a la filosofía de la programación estructurada.

Reestructuración de datos

Análisis del código fuente, en primer lugar se evaluarán todas las sentencias del lenguaje de programación con definiciones de datos, descripciones de archivos, de E/S, y descripciones de interfaz. Esta actividad a veces se denomina análisis de datos.

Una vez finalizado el análisis de datos, comienza el rediseño de datos. En su forma más sencilla se emplea un paso de estandarización de rediseño de datos que clarifica las definiciones de datos para lograr una consistencia entre nombres de objetos de datos, o entre formatos de registros físicos en el seno de la estructura de datos o formato de archivos existentes. Otra forma de rediseño, denominada racionalización de nombres de datos, garantiza que todas las convenciones de denominación de datos se ajusten a los estándares locales, y que se eliminen las irregularidades a medida que los datos fluyen por el sistema.

Cuando la reestructuración sobrepasa la estandarización y la racionalización, se efectúan modificaciones físicas en las estructuras de datos ya existentes con objeto de hacer que el diseño de datos sea más efectivo. Esto puede significar una conversión de un formato de archivo a otro, o, en algunos casos, una conversión de un tipo de base de datos a otra.

¿Qué es Refactoring?

Refactoring es un tipo de reestructuración de código que se aplica en desarrollos orientados a objetos y que se define como “el proceso de cambiar el software de un sistema de manera que no altere su comportamiento externo pero mejorando su estructuración interna” (Fowler[footnote], 2000).
Fowler, M. (2000) Refactoring: Improving the design of existing code. Addison-Wesley.Página de M.Fowler sobre refactoring y catálogo de técnicas: http://www.refactoring.com/

Como cualquier reestructuración de código, el refactoring tiene como objetivo limpiar el código para que sea más fácil de entender y modificar. Esta acción consigue, como efecto lateral, mejorar el diseño del software y ayudar a encontrar errores ocultos que puede que no hayan salido a la luz todavía.

Existen una enorme variedad de técnicas para hacer refactoring (Fowler, 2000), que pueden agruparse en las siguientes categorías:

1. (Re)Composición de métodos. En esta categoría se incluyen las técnicas para que se basan en acortar métodos demasiado largos (de acuerdo a la funcionalidad), sustituir llamadas a los métodos por su código, etc.
2. Movimiento de características entre clases. Bajo esta categoría se incluyen técnicas para reasignar las responsabilidades de una clase a otra, por ejemplo, separar una clase en varias, eliminar una clase, introducir uno o más nuevos métodos a una clase, etc.
3. Reorganización de datos. Esta categoría incluye las técnicas de refactoring que permiten facilitar el trabajo con datos, como la creación de accesotes para consultar los propios atributos dentro de una clase, reemplazar ciertas estructuras de datos por objetos, reemplazar literales por constantes, etc.
4. Simplificación de expresiones condicionales. Esta familia de técnicas pretende facilitar la comprensión, depuración y mantenimiento del software mediante la simplificación de las estructuras condicionales, por ejemplo dividiendo una condicional compleja en varias,

eliminando expresiones condicionales redundantes en estructuras complejas, etc.
5. Simplificación de las llamadas a los métodos. Las técnicas de esta categoría pretenden simplificar la interfaz de una clase para que sea más fácil de usar. Incluye algunos refactorings como cambiar el nombre de métodos, eliminar o añadir parámetros a las signaturas de los métodos, etc.

Reorganización de la jerarquía generalización. Bajo esta categoría se engloban las técnicas que permiten mover métodos a lo largo de la jerarquía de herencia, como añadir un método de subclases a una superclase, añadir constructores a las superclases o dotar su cuerpo de mayor funcionalidad, crear nuevas sub/superclases, etc.

Métafora de los dos sombreros

Lo ideal es hacer el refactoring sobre la marcha, según se va escribiendo código. La idea la explica perfectamente la metáfora de los dos sombreros. Según esta metáfora, un programador tiene a su disposición dos sombreros. Uno de ellos etiquetado "hacer código nuevo", y el otro con la etiqueta "arreglar código".

Cuando empieza a programar, se pone el sombrero de "hacer código nuevo". Se pone a programar hasta que tiene hecha alguna parte del programa y le hace una primera prueba, compilando y viendo que funciona. Deja esa parte del programa funcionando. Sigue con el mismo sombrero puesto y se prepara para seguir haciendo su programa. En ese momento ve un trozo de código que podría reaprovechar si estuviera separado en otra función o ve cualquier otra cosa que si estuviera hecha de otra forma, le facilitaría la tarea de seguir con su programa. O simplemente ve algo que no le convence cómo está hecho. En ese momento se cambia el sombrero. Se quita el de "hacer código nuevo" y se pone el de "arreglar código".

Ahora sólo está arreglando el código. No mete nada nuevo. Echa un rato cambiando código de sitio, haciendo métodos más pequeños, etc, etc. Al final deja el código funcionando exactamente igual que antes, pero hecho

de otra manera que le facilita seguir con su trabajo. Nuevamente se cambia el sombrero por el de "hacer código nuevo" y sigue programando.

La idea es hacer código nuevo a ratos, arreglar código existente a ratos. Tener claramente qué se está haciendo en cada momento. Si añadimos código nuevo, NO arreglamos el existente. Si estamos arreglando el existente, NO añadimos funcionalidades nuevas. La idea también es arreglar el código con frecuencia, cada vez que veamos que algo no está todo lo bien hecho que debiera. Es decir, hacer refactoring sistemáticamente.

Ventajas de hacer Refactoring

Cualquier programador con un poco de experiencia sabe que nunca se diseña bien el código a la primera, que nunca nos dicen al principio todo lo que tiene que hacer el código, que nuestros jefes, según vamos programando y van viendo el resultado van pidiendo cosas nuevas o midificaciones, etc.

El resultado de esto es que nuestro código, al principio, puede ser muy limpio y estar bien organizado, siguiendo un diseño más o menos claro. Pero según añadimos cosas y modificaciones, cada vez se va "liando" más, cada vez se entiende pero y cada vez nos cuesta más depurar errores.

Si vamos haciendo refactoring sistemáticamente cada vez que veamos código feo, el código se mantiene más elegante y más sencillo. En fases más avanzadas de nuestro programa, seguirá siendo un código legible y fácil de modificar o de añadirle cosas.

Aunque inicialmente parece una pérdida de tiempo arreglar el código, al final del mismo se gana dicho tiempo. Las modificaciones y añadido tardan menos y se pierde mucho menos tiempo en depurar y entender el código.